Bidding Kit 2.0 - Technical Documentation

Introduction

CONTEXT

FB focuses on optimising publisher value for in-house bidding publishers and Bidding Kit 2.0 is designed to achieve this with the goal of showing significant ARPDAU lift

WHAT IS BIDDING KIT 2.0

Bidding Kit is a lightweight SDK that enables publishers using own in-house mediation solutions to enable bidding and improve yield. It comprises of:

- Fair auction - a first-price auction where every bidder gets a fair chance to win impressions
- Adapters to various bidders - to enable bidders from a range of companies
- AB-testing to measure the incremental benefit of bidding to the publisher
- Well-defined interfaces between auction and traditional waterfall

WHY IS IT IMPORTANT?

We identified a few key issues challenging in-house publishers as they move to bidding. Bidding Kit 2.0 was designed to address the following pain points:

- Demand density - publishers are often connected to 1-2 real-time bidders. This directly affects publisher yield (measured as ARPDAU from ads), because yield is increased in dense, competitive auctions.
- Onboarding - Publishers are now required to implement the auction themselves and to integrate with various real-time bidders find it challenging. More specifically, unifying the two concepts of auction and waterfall is slowing them down.
- Scale - In-house is difficult to scale without an off the shelf solution because each integration requires a high degree of customisation which makes them costly to support.

WHO IS IT FOR?

Bidding Kit is intended for in-house publishers using home-grown mediation system (as opposed to publishers using Google, MoPub, Fyber, etc.). First version of bidding kit targets client-side mediation systems.

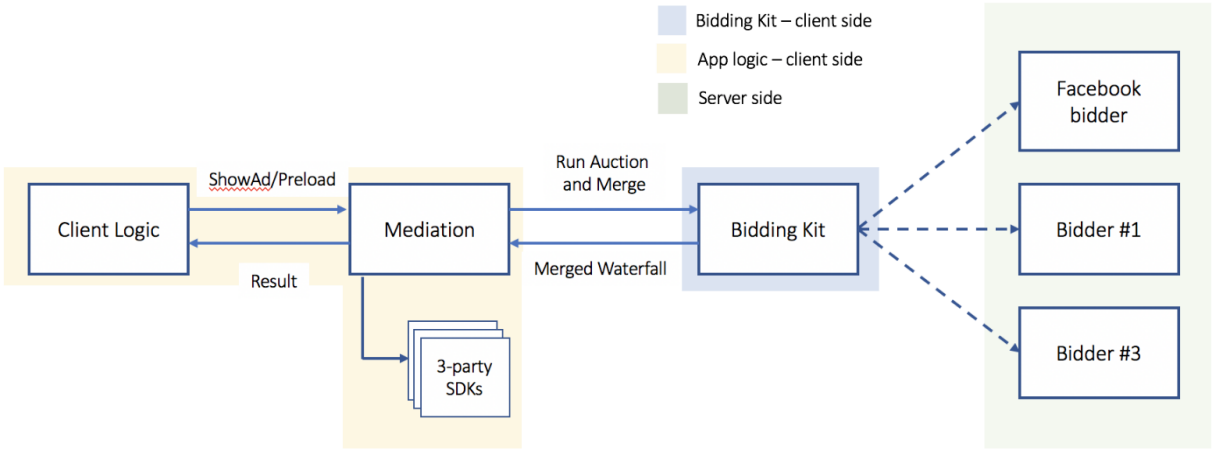
Definition: A mediation solution is client-side if **the end device consolidates information and makes the decision as to what provider will win the impression**. This distinction is important for implementation, and since no solution is either purely server or client-side.

Product Description

Bidding Kit implements a fair client-side auction, allowing connected demand sources to bid in-real time to win impressions. Default implementation relies on ORTB, but other protocols could be supported in the future. It relies on SDK integrations for ad rendering and signal collection. This model enables demand sources to use their own technology and retain direct relationships with publishers.

ARCHITECTURE

Bidding Kit runs the auction and acts as a bridge to the current mediation systems. In contrary to mediation solutions such as MoPub and AdMob, it does not manage ad loading and display logic, and leaves them to the publisher's existing system to handle. The figure below illustrates the concept in high-level:



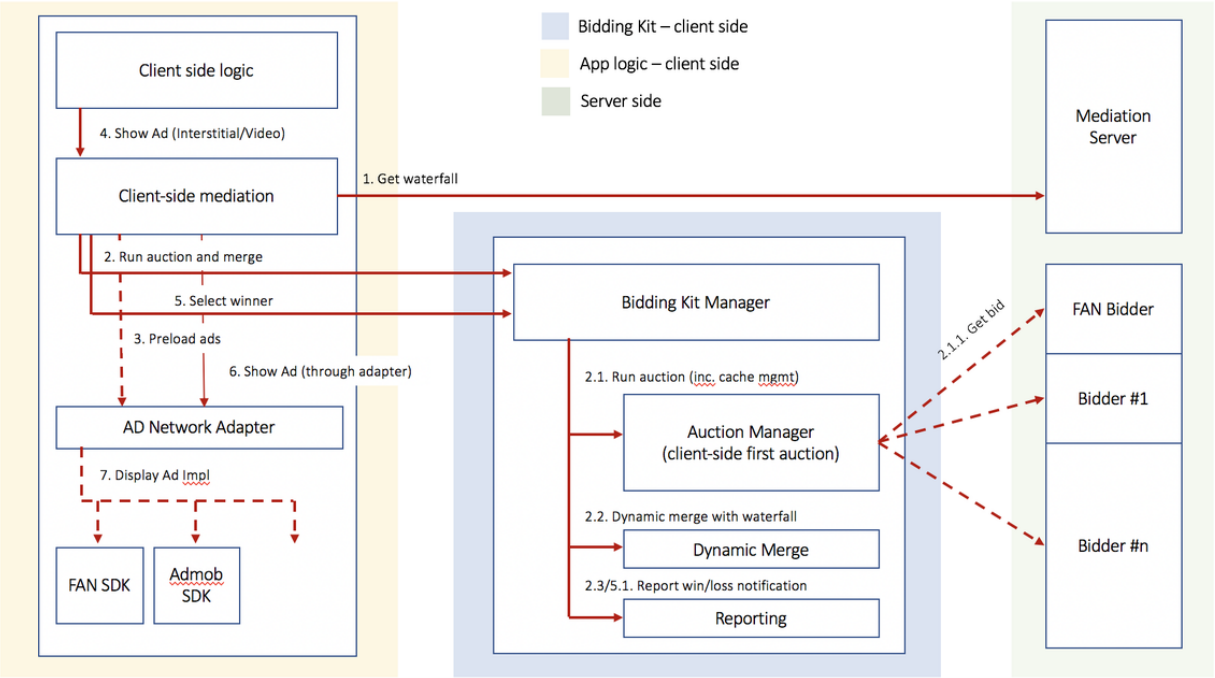
In practice, the flow is slightly more complex as Bidding Kit has to integrate with certain aspects of the publisher's mediation. See ⚓System Flow for further details.

SYSTEM FLOW

Traditional mediation systems normally comprise of three separated concerns:

- Waterfall management - according to historical averages
- Ad load - balancing between resources, UI and yield
- Ad display - simple abstraction on top of ad SDKs

The figure below illustrates a standard flow of how a mediation system is integrated with Bidding Kit:



The basic flow consists of the following steps:

1. Upon launch - the mediation system retrieves waterfall config from server (standard implementation)
2. Client-side mediation calls to our bidding kit manager to run auction logic and dynamically merge results with waterfall (**note:** Auction Manger, Dynamic Merge, and reporting are internal classes and are used here for illustration purposes)
    a. Auction Manager - A first-price client-side auction runs, and a winner is chosen
    b. Dynamic merge - used as the bridge between bidding and the waterfall system, dynamically updates waterfall structure with real-time bids information
    c. Reporting - auction result is reported including clearing price
3. Ads are loaded in the background by the mediation system
4. Client logic attempts to display an ad (e.g. interstitial/video/etc.)
5. The mediation selects the demand sources to display the ad based on price and availability
    a. Reporting - the final winner is reported to all demand sources
6. The mediation selects the right adapter and an ad is shown
7. An ad is displayed and rendered by the appropriate SDK

Integration

Setting up Bidding Kit 2.0 comprises of the following steps

1. Implement **Waterfall** and **WaterfallEntry** - Bidding Kit 2.0 requires reading and interacting with the waterfall, and it does so through these two interfaces
2. Implement **ABTest** and **ABTestSegment** - to enable AB-testing support
3. Initialise the system using **Bidder** and **Auction** - to define the different auctions per segment
4. Run the auction - when the app is launched and after the waterfall is updated on client side
5. Pre-load ads - as you would normally do using the updated waterfall retrieved once the auction has finished

COMPONENTS

Waterfall and Waterfall entry

This product does not provide mediation platform. However, it does need to interact with the publisher's waterfall setup. A Waterfall represents the ordered list of traditional waterfall line items based on their historical eCPMs. We call each entry as WaterfallEntry. A simple waterfall may look something like this:

Network	eCPM
A	$10
B	$9
A	$8
C	$7
B	$6

Each entry has the network information (e.g. name) and the eCPM either historical average or real time. We should have the ability to modify the waterfall and insert/delete entries from the list. We'll add new entries in the waterfall for each bidder. The final waterfall will still look similar - different entries ordered by eCPM except that it'll also contain the bidder entries with real-time price.

Network	eCPM
A	$10
FB Bidder	$9.50
B	$9
Bidder1	$8.50
A	$8
C	$7
Bidder2	$6.20
B	$6

Bidder

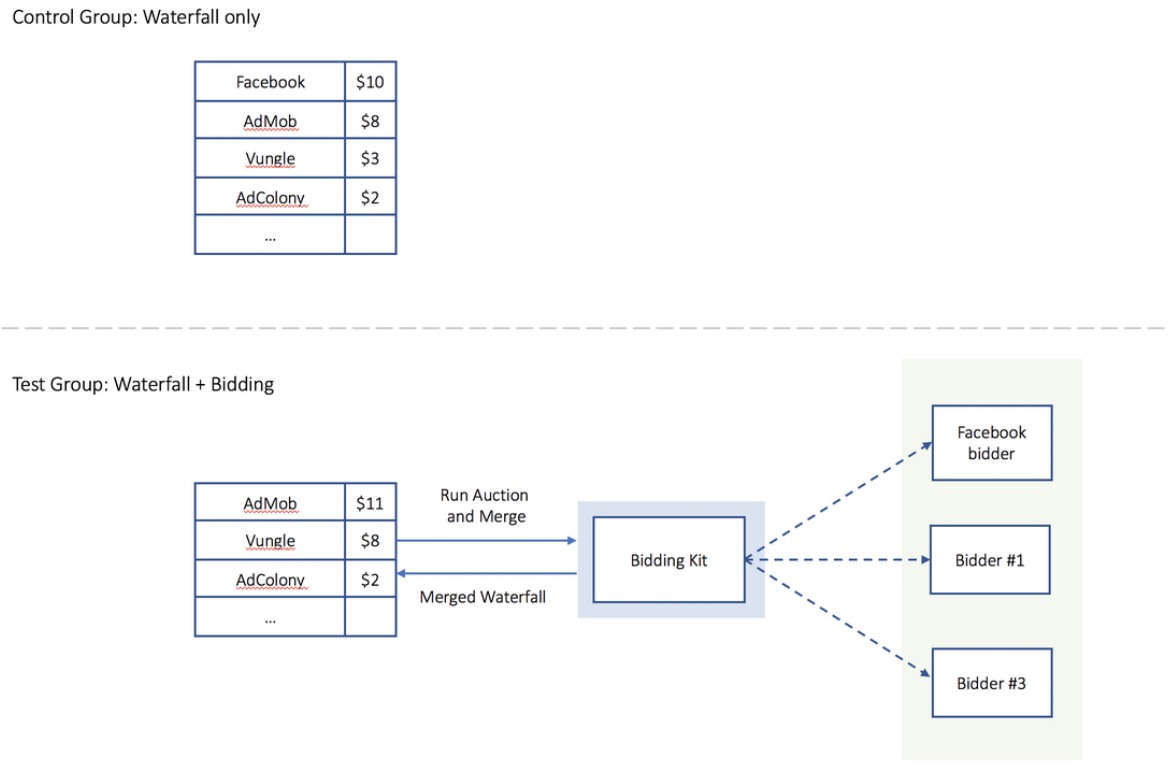
A bidder represents a real time bidding ad network e.g. Facebook bidder. It has the ability to give real-time price for an ad opportunity. Bidding Kit 2.0 will integrate with FB and few other bidders. The publisher need not worry about sending requests to the bidders, it will be done by our SDK.

Auction

Our SDK runs an auction among the real time bidders. Once we've updated the waterfall with the new entries, the publisher should pick the waterfall entry that fills with the highest price, just like conventional waterfall setup. This logic should be implemented by the publishers. The publishers should also let us know, through the API call, which entry was selected as the winner before displaying the ad.

A/B Test

Bidding Kit 2.0 will provide the ability to A/B test to check the incremental gains by adding new bidders to the current setup. For this, we the publishers should have the ability to equally split their users into multiple segments so that each user should belong to the same segment for each request till the time of experiment. Our SDK provides the ability to have different setups of bidders for different segments. Here's a diagram showing an A/B test setup between waterfall and 3 new bidders using bidding kit 2.0.



ANDROID

1. Waterfall & WaterfallEntry.

You will have to implement this interfaces.

- **WaterfallEntry**
  - The WaterfallEntry contains an entry in your Waterfall.
    - The bid is the bid returned for the given bidder
      - It can be null if this entry wasn't made from a bidder.
    - The bidderName is an Identifier for the given entry
    - The CPMCents is the historical value for your CPM or the bidding returned value.

```
public interface WaterfallEntry {
    // return the bid for this waterfall entry, if it exists
    Bid getBid();
    // return the CPM in cents for this waterfall entry
    double getCPMCents();
    // return the bidder name for this waterfall entry
```

```
    String getBidderName();
}
```

- **Waterfall**
    - contains a list of Waterfall entries, sorted by CPMCents.

```
public interface Waterfall {
    // creates a WaterFallEntry with the given bid and inserts it into
    // the waterfall
    void insertFromBid(Bid bid);
    // inserts the given WaterfallEntry into the Waterfall
    void insert(WaterfallEntry waterfallEntry);
    // returns an Iterable of WaterfallEntry for this Waterfall
    Iterable<WaterfallEntry> entries();
}
```

2. ABTest & ABTestSegment

- You need to implement these interfaces in order to be able to do ABTesting

```
public interface ABTestSegment {
    // will return an identifier for the experiment
    String getSegment();
}
```

```
public interface ABTest {
    // equally splits the users into different segments
    ABTestSegment getCurrent();
}
```

- Example
    - You will have to implement these interfaces yourself. They are not part of the API
    - Create the segments and add them to the ABTest

```
        ABTest abTest = new ABTestImpl();
        ABTestSegment testA = new ABTestSegmentImpl();
        ABTestSegment control = new ABTestSegmentImpl();
        // the addSegment method is not mandatory and not part of our API
        abTest.addSegment(testA);
        abTest.addSegment(control);
```

3. Bidder

```
Bidder facebookBidder = new FacebookBidder(
    // the context of your given application
    context,
    // Your app id
    "MY_APP_ID",
    // Your placement ID given by MOMA
```

```
    "MY_PLACEMENT",
    FBAdBidFormat.BANNER_320_50));
```

4. Auction

- This is an object to which you will pass all the bidders and segments you want the bidders to belong to.

```
// Create an auction using ABAuction.Builder
ABAuction auction = new ABAuction.Builder()
                // You can add multiple bidders to a certain segment
                // we will choose the bidders to run based on what
                // ABTest.getCurrent() will return
                .addBidder(testA, facebookBidder)
                // For a control group you can use addSegment, which will
                // require no bidders
                .addSegment(control)
                .build();
```

- Implement the **AuctionListener** class

```
public interface AuctionListener {
    // This method will be called when we populated the waterfall with all
    // entries and finished bidding.
    // The waterfall parameter is a copy of your original waterfall with
    // the added bidding waterfall entries
    //
    // In the method you should
    // 1. Do auction.notifyDisplayWinner(mAbTest, displayWinnerBid);
    //   a. You should call this before showing the ad.
    //   b. We need this in order to know who won the bid and is going
    //      to be shown
    // 2. Use the waterfall to show the desired ad.
    void onAuctionCompleted(Waterfall waterfall);
}
```

- Call **startABAuction** on the **ABAuction** or **Auction** instance

```
// startABAuction
// **In this method we will populate the waterfall with the bids
// received from the bidders you added to the auction.
// abTest
// **The bidders will be chosen based upon the segment returned
//   (abTest.getCurrent())
// AuctionListenerImpl
// **When we finished calling all the bidders and populating the waterfall we
// will call onAuctionCompleted from AuctionListenerImpl
auction.startABAuction(
                // A instance of the ABTest class you implemented
                abTest,
                // A instance of the Waterfall class you implemented
                waterfall,
```

```
                // A instance of the AuctionListener you implemented
                new AuctionListenerImpl());
```

5. Sample Code

```
        // 1. Create the waterfall and add all the entries we have
        waterfall = new WaterfallImpl();
        mWaterfall.insert(new WaterfallEntryImpl("NetworkA", 5));
        mWaterfall.insert(new WaterfallEntryImpl("NetworkB", 7));
        mWaterfall.insert(new WaterfallEntryImpl("NetworkC", 3));

        // 2. Create the AbTest
        AbTest abtest = new ABTestImpl();
        ABTestSegment testA = new ABTestSegmentImpl();
        ABTestSegment control = new ABTestSegmentImpl();
        abTest.addSegment(testA);
        abTest.addSegment(control);

        // 3. Create the Facebook bidder
        Bidder facebookBidder = new FacebookBidder(
            context,
            "MY_APP_ID",
            "MY_PLACEMENT",
            FBAdBidFormat.BANNER_320_50));
        // 4. Create an auction
        auction = new ABAuction.Builder()
                .addBidder(testA, facebookBidder) // What Segment the bidder is in
                .addSegment(control) // a control segment with no bidders
                .build();
```

```
        // 4. Implement AuctionListenerImpl and call startABAuction
        auction.startABAuction(
                abTest,
                waterfall,
                new AuctionListenerImpl() {
                    @Override
                    public void onAuctionCompleted(Waterfall resultWaterfall) {
                        // 1. Override the current waterfall with new one
                        // 2. Start pre-loading ads
                    }
                });
```

IOS

1. Waterfall & Waterfall Entry

Implement the following protocols for waterfall logic using waterfall entry as abstract entity abstraction.

```
@protocol FBBKWaterfall <NSObject>
// returns an waterfall entry at specific index or nil if index out of bounds
- (id<FBBKWaterfallEntry>)entryAtIndex:(NSUInteger)index;
// number of entries in waterfall
- (NSUInteger)entriesCount;
```

```
// creates waterfall entry from bid, inserts and sort to keep order
- (void)insertEntryUsingBid:(FBBKBid *)bid;
// creates waterfall entry from bidder and cpm in cents, inserts and sort to keep orde
- (void)insert:(NSString *)bidderName cpm:(double)cpm;
@end
```

```
@protocol FBBKWaterfallEntry <NSObject>
// provides specific bid if available or nil
@property (nonatomic, readonly) FBBKBid *bid;
// returns cpm in cents
@property (nonatomic, readonly) double CPMCents;
// returns name of bidder
@property (nonatomic, readonly) NSString *bidderName;
@end
```

Create waterfall and provide historical cpm

```
id<FBBKWaterfall> waterfall = [[Waterfall alloc] init];
[waterfall insert:@"NetworkA" cpm:5];
[waterfall insert:@"NetworkB" cpm:7];
```

2. ABTest & ABTestSegment

Implement A/B testing supporting protocols.

```
@protocol FBBKABTest <NSObject>
// splits users into pieces
@property (nonatomic, readonly) id<FBBKABSegment> currentSegment;
@end
```

```
@protocol FBBKABSegment <NSObject>
// identifier of a segment
@property (nonatomic, readonly) NSString *segment;
@end
```

Create A/B Segments and Tests

```
id<FBBKABTest> abtest = [[ABTest alloc] init];
id<FBBKABSegment> testA = [[ABSegment alloc] init];
id<FBBKABSegment> control = [[ABSegment alloc] init];
[abtest addSegment:testA];
[abtest addSegment:control];
```

3. Bidder

Implement bidders or use predefined confirming to protocol.

```
@protocol FBBKBidder <NSObject>
// provide bid asynchronously. will invoke one of two callbacks
// 'complete' if bid retrieving is succesful, providing bid itself and cpm in cents
// 'failed' otherwise, will contain error object describing what happened
```

```
- (void)retrieveBidComplete:(void (^)(FBBKBid *bid, double cpm))complete
                     failed:(void (^)(BOOL cancelled, NSError *error))failed;
@end
```

Setup bidders that you going to use.

```
id<FBBKBidder> bidder = [[FBBKFacebookBidder alloc] init];
```

4. Auction

Protocol for A/B auction

```
@protocol FBBKABAuction <NSObject>
// delegate for providing additional strategy
@property (nonatomic, weak) id<FBBKABAuctionDelegate> delegate;
// starts auction. provide test and waterwall
- (void)startWithABTest:(id<FBBKABTest>)test waterfall:(id<FBBKWaterfall>)waterfall;
// notify about waterfall entry that wins
- (void)notifyWinnerEntry:(id<FBBKWaterfallEntry>)winner;
@end

// this protocol represents the notifier object. do any logic by implementing specific
@protocol FBBKABAuctionDelegate <NSObject>
@optional
- (void)abAuctionDidComplete:(id<FBBKABAuction>)abAuction winnerEntry:(id<FBBKWaterfal
@end
```

Create auction and pass bidders

```
id<FBBKBidder> facebookBidder = [FBBKFacebookBidder new];
FBBKABAuctionImpl *auction = [[FBBKABAuctionImpl alloc] init];
// provide any additional bidders you want to use
[auction addBidder:facebookBidder forSegment:testA];
```

Setup delegate

```
auction.delegate = self; // become a delegate to handle callbacks
```

```
- (void)abAuctionDidComplete:(id<FBBKABAuction>)abAuction winnerEntry:(id<FBBKWaterfal
{
    // TODO: log winner
}
```

Start auction and pass test & waterfall

```
[auction startWithABTest:abtest waterfall:waterfall];
```

5. Sample Code

```
- (void)createAuction
{
```

```
    // create and setup waterfall
    id<FBBKWaterfall> waterfall = [[Waterfall alloc] init];
    [waterfall insert:@"NetworkA" cpm:5];
    [waterfall insert:@"NetworkB" cpm:7];

    // setup A/B testing
    id<FBBKABTest> abtest = [[ABTest alloc] init];
    id<FBBKABSegment> testA = [[ABSegment alloc] init];
    id<FBBKABSegment> control = [[ABSegment alloc] init];
    [abtest addSegment:testA];
    [abtest addSegment:control];

    // setup bidders and auction
    id<FBBKBidder> bidder = [[FBBKFacebookBidder alloc] init];
    FBBKABAuctionImpl *auction = [[FBBKABAuctionImpl alloc] init];
    [auction addBidder:facebookBidder forSegment:testA];
    auction.delegate = self;
    [auction startWithABTest:abtest waterfall:waterfall];
}

- (void)abAuctionDidComplete:(id<FBBKABAuction>)abAuction winnerEntry:(id<FBBKWaterfal
{
    // 1. Override the current waterfall with new one
    // 2. Start pre-loading ads
}
```